Final Report

Department of Computer Science

Calvin University

LoTide: An Adaptive Music Workflow

Mark Wissink, Matthew Nykamp, Ian Park

Date: 4/26/20

Mentor: Joel Adams

Honors Project: No

https://cs.calvin.edu/courses/cs/396/private/final.html

Project Vision and Overview

Video games products and similar media have been a driving force behind many advances in technology. Graphics cards, virtual reality, and powerful software are continually becoming more powerful to accommodate a better experience of gaming. However, the influence of video games has also extended to other media such as art and music. Adaptive music is a recent paradigm of composition that accounts for the non-linearity of video games. Using techniques like horizontal scaling, a song can slowly transition into another song; and vertical techniques can reorchestrate the current song. These techniques make for a more immersive experience as the soundtrack adapts to a player's surroundings.

Creating adaptive music is a non-trivial problem, and existing products provide software solutions to compose this type of music for games. However, most of these existing solutions emphasize horizontal techniques, shifting between pre-rendered layers at runtime. There seems to be a small disconnect between the digital audio workspace (DAW) and the runtime API for adapting the music. Many vertical techniques suffer from this disconnect because pre-rendered audio does not provide that flexibility. The vision behind Lotide is to provide even more flexibility with audio in nonlinear use cases by providing an interface to a DAW that plays music in real-time. By providing a cross-platform audio backend, developers will be able to interface through a language agnostic API to manipulate and play music in their program. Another important aspect to our project was the licensing that is associated with our software. Most existing solutions have proprietary licensing and are not open-source. Our vision is a FOSS (Free and open-source software) solution to adaptive music.

Background

Nonlinear media usually takes form in the shape of video games. By nonlinear media, we refer to the type of media which is not necessarily static, as a movie soundtrack would be, but rather an auditory experience which can be different depending on external variables. This is commonly called adaptive music. For example, in the final lap of Mario Kart, the music is pitch-shifted to a higher than normal value and the tempo is increased, resulting in a more suspenseful overall feel. In this way, the variable lastLap (as it might be implemented in Mario Kart) determines what type of music is played.

Another example is sound effects. Many games have sound effects which are played when a specific action or event occurs. Attacking with your character might play an accompanying sound of a sword whirring through the air, or moving the character might produce a footstep sound. In this way, the overall auditory experience is changed when some external factors change.

One downside to these existing methods of implementing adaptive music is that blending different segments of rendered audio can be jarring - this can be seen in the Mario Kart example. When you cross the finish line, the sound immediately jumps to a small transitional sound effect, and then immediately jumps back to the pitch shifted sound, probably to avoid some form of auditory artifact caused by either an immediate jump to the pitch shifting or a slower ramping up of the pitch and tempo.

Another downside of existing solutions is that they scale poorly. If a developer wishes to have a soundtrack that reacts to the game—by introducing a new instrument on top of the existing sound, that would be easily manageable by creating a set of rendered audio for each instrument, and then blending the new audio in. However, having a different segment of rendered audio for each instrument in a song quickly becomes unmanageable as the number of instruments increases, not only in terms of performance (playing 25 or more rendered audio at the same time could cause problems), but also in terms of managing which should be playing or not be playing at any given time.

We decided early on that many sound effects in games should be immediate, and probably don't need to be involved in the music of the game. In both examples earlier, having the sound effect play immediately is important. However, our system would support a sound effect being played in time with the music, if appropriate. But, we didn't design with this in mind - our focus is mostly on providing a way to create music.

The essential problem seemed to be that the parts that made up the music of many games was inaccessible to programmers of the game - all they had access to was rendered audio. They overcame the limitations of rendered audio in a number of ways, as we shall see shortly, but we decided that creating a system which allows a programmer control over these elements would allow for even more possibilities. Having a way to easily add or remove certain musical layers, either horizontally (changing what notes the instruments might be playing) or vertically (changing which instruments are playing entirely, or switching a set of notes from one instrument to another) seemed like the way to go.

So, we decided that we should begin looking at tools that game developers have been using to satisfy these problems in order to figure out what exactly our proposed workflow would look like, and if it was really needed. From our research, we found FMOD and Elias to be the popular options for adaptive music. With plugin support on industry standards like Unity and Unreal, these programs had a lot of offer. The innovation that LoTide makes on these products is real-time audio rendering. With this method, users are able to manipulate composition data at runtime. Unlike FMOD and Elias which primarily use pre-rendered audio, LoTide allows users to rearrange notes to change tempo, match key, and generate notes on the fly. Another factor to consider is that both softwares, FMOD and Elias, are proprietary, closed source software. When looking for FOSS solutions, we came across OAML (Open Adaptive Music Library). Similar to FMOD and Elias, the solution to dynamic in OAML is to compose music out of pre-rendered loops. While this solution does work well, our goal with LoTide is to take adaptive music a step further with dynamic sound and composition at runtime.

Implementation and Design

TSAL (Thead Safe Audio Library)

Why TSAL?

TSAL is a project created by Joel Adams and Mark Wissink to complement the existing TSGL (Thead Safe Graphics Library); both are tools that can be used in the instruction of the parallel computing paradigm. TSAL provides a minimal implementation of a DAW that can be used to generate audio given notes to play. TSAL was a natural choice as a backend for LoTide since it both inspired and fit the specifications for the project. Most importantly, TSAL provides a simple, lightweight DAW that can do real-time audio generation which supports the goals of the adaptive music framework. Also, using TSAL in a full-fledge project was a good test of the functionality that TSAL provided, and it gave insights into what needed to be improved.

Audio Backend

TSAL had originally used RtAudio as it's solution to cross-platform audio library. While RtAudio worked well, it was primarily chosen as it was easy to incorporate into the build system early on. However, RtAudio is not actively developed and lacks good documentation. In its place, PortAudio was chosen since it is actively developed and used as a solution in many other projects. The audio library has always been tightly coupled to the TSAL Mixer class, and switching between the libraries was trivial in the code. However, successfully adding PortAudio to the build system was a little more complex.

Another issue in the original development of TSAL was the lack of support for stereo (2+ channel) audio. Initially developing with mono (1 channel) audio made the project more maintainable, but it was an oversight in future uses of the library. With switching the audio backend, it was a natural time to add stereo support to the library. To handle the logic of stereo audio, the AudioBuffer class was created. AudioBuffer handles the logic of multiple channels and frames. It also handles the logic behind interleaving. Interleaved audio stores audio samples in a single buffer grouped by frame. In a stereo example, the first two values in the buffer are the left and right channel samples for the first frame, the next two values being the left and right channel samples for the second frame, and so on. Rather than having multiple buffers in memory representing each channel, interleaved audio takes advantage of the spatial locality in the CPU cache since audio frames are generally accessed sequentially.

Thread Safety

The thread safety functionality is a core part of TSAL since it's goal is to provide a library that can easily handle complexities of multi-threading. The initial approach to solve this problem was for each device to provide thread-safe operations for adding new devices. If the model of TSAL can be thought of as a graph with the Mixer at the root, each node has to provide thread-safe operations for adding new nodes. The main problem with this implementation is that it can cause a buffer underrun in the underlying audio output. For example, if the audio library requests new samples from the Mixer class, it could fail to receive samples in time in the case that some Channel on the Mixer is locked because of adding new devices. The solution implemented was to have a single lock on the model that would ensure changes to the model where made after audio had been generated for the current audio request. Now Channels and other devices that route audio can make asynchronous requests to the Mixer to make changes to the model.

Plugins

One the goals for LoTide and TSAL was to have plugin support for virtual instruments and effects. This would give developers access to a wide array of different sounds rather than just the native TSAL instruments. Some time was dedicated to researching what type of plugins could be supported—most being some form of dynamic libraries. One consideration we had to make in the case of LoTide was the licensing on the software plugin. Since LoTide generates audio at runtime, whatever plugins used to generate the audio would have to be bundled with the project file. And since these files are expected to be distributed to many users, this could potentially be a violation of many redistribution clauses on proprietary plugins. Once again, we turned to the FOSS solution of LADSPA (Linux Audio Developer's Simple Plugin API) and LV2 (LADSPA Version 2). However, loading dynamic libraries in a cross-platform environment proved to be a difficult task. A successful implementation was made for Linux but not for Windows.

Synthesizer Improvements

As an alternative to the incompleted plugin support, we focused on extending the functionality of the PolySynth synthesizer in TSAL. At the start of the project, PolySynth only supported sine, saw, and square waveform outputs with an amplitude envelope. To make PolySynth robust, many new features where added including white noise, filters, phase modulation, frequency modulation, and LFO (low frequency oscillator). White noise was an addition to the Oscillator class which generates random values as output.

The Filter class implements an algorithm that works as a low-, high-, and band-pass filter. The phase and frequency modulation where more additions to the Oscillator that manipulate the output waveform by some given input, usually another waveform. The LFO is simply an oscillator that can be used to slowy modify some value such as a volume or frequency cutoff over time, making the sound more alive.

Device Parameters

While developing the frontend of LoTide, it became clear that the way parameters where being handled on devices could be improved. Before, each parameter had its own setter and getter. While this works fine in code, it can be cumbersome when designing interfaces for these systems. An index based parameter system, similar to the LADSPA plugins, seemed to be a good solution. ParameterManager is the implementation of that system, providing the necessary methods to define and access parameters on devices. By using enums as an alias for the index, programmers can easily read what parameters they are modifying and safely make changes in code.

Lotide Backend

While we had TSAL as a solid base for generating the audio, we realized quickly that we would need many structures forming a layer of abstraction in order to create a usable library. While TSAL has a PolySynth, for example, we needed to provide a way for a user to specify what a song should look like without manually managing the calls to play and stop for each note.

In essence, we wanted the user to be able to specify a song, which we ended up thinking of as a collection of instruments playing some collection of notes. We wanted the user to be able to not only start or stop a piece of music, but interact with it in more ways. To facilitate this, notes were stored in collections known as phrases. Under the new schema, the song would be composed of a number of phrases, where each instrument plays some phrase at a given time.

As our music is adaptive, we required additional information regarding which phrase each instrument would play given the current time. To do this, we created a grouping, which simply maps each instrument to a list of phrases to play. By creating new groupings and telling the song to switch to a new group, the user can now create almost any type of non-linear music at the behest of outside inputs.

The last piece of the puzzle comes in the form of a sequencer, which periodically asks the song which notes should be playing. It plays those notes by telling the TSAL instrument to play it. It remembers which notes are playing, and when they started, so it can at the appropriate time tell the synthesizer to stop playing the notes. When the main lotide object is told to start playing, it creates a new thread to facilitate this, allowing for other lotide interactions to occur simultaneously to the playing of a song.

With all of these structures in place, it is now easy for a user to simply swap out what is being played by certain instruments at runtime - a single function call in the game code will do the trick. In addition, to define new composition at runtime is possible, and some other small benefits present themselves as well. For instance, the ability to scale our music's pitch up or down is trivial, and presents no artifacts as rendered audio would.

One potential issue with our system is that the complexity of instruments and effects is limited to what can be processed in real time. Considering that some professionally created music commonly takes render times on the order of minutes, these types of sounds may be impossible in our real time context, and would also be user-dependent (so a song that would work well on a really nice computer might work poorly on a low-spec one). Discovering the bounds of our system is definitely an important area to explore should Lotide be proven useful. Given that our target audience is primarily indie game developers who lack the budget for other commercial grade sound software, this may be an acceptable downside.

Cross-Platform

The Build System

In order to obtain cross platform support, we had to use a build system for simplifying the build process and detecting the platform. The main branch of TSAL was using a workflow that incorporated the traditional GNU build tools. This worked well in UNIX environments but required MinGW to work on Windows. These tools were not as reliable and straightforward as the official build tools offered by Microsoft through Visual Studio. We also had to decide between using a compatibility layer on top of the operating system or writing portable C++ code and compile natively on our target platforms. While using a compatibility layer would have made the code less complex, it would come at a performance cost. Since LoTide and TSAL would be running in a real-time system where execution time is important, we decided against a compatibility layer in favor of a native compilation approach. So, we ported the project to CMake, which offered build outputs targeting the Windows compiler and GCC on unix systems.

We tried to utilize many modern CMake features such as embedding external library dependencies as git submodules and automatically generating packages for Linux distributions, namely Debian(deb) and RedHat(rpm). Furthermore, we made it so that our library would also be easily consumed by other modern CMake projects just by invoking an `add_subdirectory(<directory>)` to our project directory.

Although the convenience was beneficial once we got it set up, we faced a lot of small problems with getting CMake to work consistently. The downside was that the official documentation for CMake was not so clear. Likewise, examples for cross-platform implementations were lacking.

Issues with OS Versioning

One of the challenges with making the software cross platform was that some bugs were not easily reproducible even among the same Windows systems. Small changes to the version of Windows itself produced different build results. Likewise, bugs on the sound output were difficult to reproduce and debug.

Serialization and File Format

Initially, we thought that our system would support Midi files and Soundfonts to incorporate them into our platform. So, we tried to find out ways to generate a compressed file containing all of these files. However, for simplicity, we omitted support for those files and chose to serialize the state of a song and provide that as a file format. At first we attempted to use Boost for our serialization because it had a pretty straightforward way of serializing native types. However, because the Windows build was cumbersome, we switched to a more modern serialization library called cereal. Since this project was compatible with CMake, we were able to easily embed this within our project.

API

POSIX Sockets/WinSock

Along with our C++ library, we wanted another way for programmers to consume our library. This was primarily due to the fact that game development often happens in other languages such as C# and Java. However, an actual port of Lotide to all of those platforms seemed like an inordinate amount of work. So, we decided to create a standardized API through a Remote Procedure Call (RPC) server. By doing so, we could send remote procedures (i.e. function names with parameters) from a client and send back a state of the system without providing any other language abstractions.

At first, we researched some pre-existing RPC projects that we might use. However, many of them had their own protocols in place, which felt like a potential overhead for us and for the users of our API. So, we decided to start from scratch and build an RPC using TCP sockets and JSON with our own protocol. So we made a client that sent a JSON string with a UUID and the procedure specifications (command and parameters) to the server. Then, the server also sent the serialized state of the LoTide instance to the client of the specified UUID. We also tried to multi-thread the server so that it could future proof it. There were two cases of the multi threaded server. The first was during development, a developer could want a running instance of the game while creating the music in a different instance or while monitoring the instance for debugging. In this case, each client would have to share the state of the LoTide instance with another client. So a UUID would identify a client and associate it with an instance of LoTide running from a different thread. And each procedure call is run by a thread generated on initialization in a thread pool.

Clients

Previous Efforts

We began our process of building a GUI editor which was planned to look much like existing editors. While investigating which framework would be ideal for this project, we initially began with QT. Our reasoning at the time was that it was widely accepted and used in many commercial and open source projects, including LMMS. At the time, it seemed to be worth the effort to learn a brand new framework in order to create a professional looking UI.

However, as we began the exploratory process of building the UI for our project, we realized that the process of learning QT was more like learning an entirely new language rather than just using a GUI framework - we realized that if we wanted our main deliverable to be as developed as we would like, a simpler solution would be preferable.

Why Web?

We eventually decided to build our proof-of-concept prototype using web technologies because it was easy to implement our ideas quickly without much boilerplate. Furthermore, since we are targeting cross platform support, we do not have to worry about native toolkits with the web.

WebSockets and Node.js

Since we decided to build web apps for our test/prototype clients, we had to find a way to use our API through the web. The only way that our daemon could actively respond to our web app was through the Web Sockets standard. Although we could host Web Sockets directly from C++, we decided, as a proof-of-concept, to use Node.js as the primary Web Sockets server to show that we could run our app using various languages that support sockets. Node.js had socket support built into the language. So, it was very convenient to make it work with our socket implementation. In addition to our prototype, we have CLI client implementations to test each layer in our protocol.

Problems(Future work?) with client prototype

One feature we are lacking is that we can’t really stream audio through the browser yet. Although we are using web technologies for our prototype, the daemon itself is local.

Results and Discussion

TSAL

Many improvements were made to TSAL over the course of the project. When it was initially created during the summer, some shortcuts were taken to simplify the project. Changes in the audio backend and multi-channel support have made TSAL a viable option for synthesizing audio. New structures and bug fixes in the code base made TSAL much more stable while simplifying the API. Adding LADSPA and LV2 plugins proved to be a complicated task, but it may be worth the effort if support were added in the future. Some options to make the task feasible would be to use a shared library loader such as Boost.DLL or QLibrary. Writing a cross-platform shared library loader from scratch is a possible solution, but it may be smarter to use an existing solution. Many new additions to PolySynth made the class a functional synthesizer capable of generating a wide array of noise. With additions of filter, modulation, and LFO, the virtual instrument proves to be a viable option for creating music. Finally, the ParameterManager class created an effective method for access and connecting device parameters to both interfaces and users.

LoTide

Conclusion

Future Work

The future goals of both TSAL and LoTide are closely aligned as we worked towards a fully featured DAW. Over the course of our project, we have implemented an MVP (minimum viable product) with lots of room for growth.

TSAL

The main addition to TSAL that would greatly benefit it and LoTide alike would be the support for virtual instrument and effect plugins. Expanding the collection of native effects and instruments would also be important since it gives new users more options if they don’t have access to many audio plugins. Another reason to add more native effects and instruments is to simply have more examples of implementations in the code base. With new students or developers working in the code, this would become an important aspect.

One weakness of TSAL is that the Mixer class is tightly coupled to the audio backend; it would be good to separate the Mixer and the audio backend. Similar to LMMS (Linux MultiMedia Studio), multiple audio backends can be used and selected by the user depending on what their machine supports. With this flexible method, if a better audio backend is discovered, it can simply be added as a new backend alongside any existing backends. In the context of LoTide, this feature would also be important since it could be running on many different platforms that may need multiple different audio backends to be supported.

LoTide

Acknowledgements

I would like to acknowledge Ian Park, Matthew Nykamp, and Mark Wissink.

References

LoTide

List of Lotide references

TSAL

LMMS (LInux MultiMedia Studio) https://lmms.io/

MusicDSP https://www.musicdsp.org/en/latest/

Appendixes